Lessons Learned in Adapting a Software System to a Micro Computer

ABSTRACT: A system was developed in a laboratory on a desktop computer to evaluate armor health. The system uses sensors embedded in the armor which causes the armor to vibrate. There are subtle changes in the vibration pattern if the armor has been damaged. The system uses these changes to diagnose armor health. The goal of the team was to take this application and transfer it to the field where it would be embedded in a portable system that could be readily used by soldiers.

The original application was developed on a desktop computer that had a powerful processor, 4 GB of memory and a standard operating system. The challenge was to take this application that had essentially unlimited resources (disk, memory and processor) and modify it to run on a microcontroller which has rather limited resources including no disk, no operating system, very little memory, and a much slower processor. There is no explicit general method that will work for every application, however it is hoped that the steps described below will provide a general framework for the process and some insight as to how to approach the task.

The Steps Involved in the Process

1. If the application has several main programs, start the conversion efforts with the simplest main program first.
2. Start with the source code for the application and reduce all array sizes to the minimum necessary to run the application.
3. Replace double precision data declarations with single precision data declarations wherever possible. This requires careful checking of the application to make sure that the required accuracy of the computations is maintained even with the reduced precision.
4. Remove all standard file input/output references from the source code while converting the application for the micro. Replace these references with appropriate workarounds.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **05 FEB 2013**	2. REPORT TYPE **Technical Report**	3. DATES COVERED **01-01-2012 to 31-12-2012**

4. TITLE AND SUBTITLE **Lessons Learned in Adapting a Software System to a Micro Computer**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) **Thomas Meitzler; Thomas Reynold; Samuel Ebenstein**	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **US Army RDECOM- TARDEC,RDTA-RTI,6501 East Eleven Mile Rd,Warren,Mi,48397-5000**	8. PERFORMING ORGANIZATION REPORT NUMBER **#23658**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) **U.S. Army TARDEC, Ground System Survivability, 6501 East Eleven Mile Rd, Warren, Mi, 48397-5000**	10. SPONSOR/MONITOR’S ACRONYM(S) **TARDEC**
	11. SPONSOR/MONITOR’S REPORT NUMBER(S) **#23658**

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
A system was developed in a laboratory on a desktop computer to evaluate armor health. The system uses sensors embedded in the armor which causes the armor to vibrate. There are subtle changes in the vibration pattern if the armor has been damaged. The system uses these changes to diagnose armor health. The goal of the team was to take this application and transfer it to the field where it would be embedded in a portable system that could be readily used by soldiers.

15. SUBJECT TERMS
microprocessor, desktop computer, algoritm modification

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Public Release**	**13**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

5. Modify the algorithms used if necessary to reduce the memory requirements so that the application will run on the micro.

The steps above will be illustrated by applying them to a practical example.

Description of the Example Application

The example we use measures armor health. Sensors are placed in armor and these sensors are subject to ultrasonic vibrations from 1 to 200 kHz by 1 kHz increments. The responses of the armor plate to these vibrations are collected, and the average and standard deviations of the strength of these vibrations are stored in a database. When the responses to the healthy armor plate are collected over time, random errors tend to cancel each other, and the means and standard deviations of the responses can be used to create a database which is called the "fingerprint database". This database represents the "ideal" behavior of the healthy armor plate and it can be used at some future time to check the armor. A data file is collected and compared to the database to determine armor health. If the data file differs substantially from the fingerprint database, it indicates that the armor has been damaged. The system was developed so that it has two main programs, the database builder and the compare function which is used to compare a new data file to the database. The main programs are quite simple, and most of the computation is done in a software library which is common to both main programs. The compare function is much simpler than the database builder, so we shall consider it first.

The Simplest Main Program: The Compare System

The compare system uses limited memory since it does a calculation using an input of 200 data points and a database of 400 data values (200 averages and 200 standard deviations). The program is written in the C programming language. The standard method for developing an application for the micro is to first develop it on a pc. The application is then downloaded to the micro for testing. The program on the pc for developing micro applications is called a cross compiler. After successful program execution on the pc, the program is downloaded to the micro and run there to test for any problems. The first step was to develop a C program on the pc that implemented the compare system without using any data i/o that would meet the micro's data memory and program memory limitations. The pc contains only one kind of memory and it can be used for either programs or data, while the micro contains two kinds of memory, SRAM for programs and flash that can be used for data.

Reduce Array Sizes

Comparison of Memory Resources Available on Micro and PC

Computer type	Flash Memory for Micro (Megabytes) For data storage	SRAM (Megabytes) For programs
Windows PC	Can use all of SRAM for programs or data (3250)	3250
Micro	0.262	0.065

Table 1: Comparison of Memory Resources

The first step was to reduce array sizes where they are much larger than actually necessary. For example if an array has dimension 1000 to accommodate 1000 files, but more than 100 files are hardly ever used, the array size can be reduced from 1000 to 100 thereby reducing the memory requirements by 90%.

Due to the fact that current desktop computers have essentially unlimited memory (a minimum of at least 4GB on most computers) programmers have become very generous with allocating array sizes. The result is that a program that uses an array of size 100 may have an array dimensioned as 1000. So the allocation statement

```
float data[1000];
```

can be replaced by

```
float data[100];
```

Replace Double Precision Declarations with Single Precision Declarations

As a general rule most mathematical computations are done in double precision to provide greater accuracy, however this is often wasteful. Most of the time float variables can be used instead of double precision, thus saving half of the space since a double precision variable is 8 bytes as opposed to 4 bytes for a float variable. The following method is a simple way of running a program in either single or double precision without major code changes. Instead of declaring floating point variables as float (single precision) or double (double precision), declare them as real, for example

```
REAL x[100],y[100];
```

where the type REAL is defined in a parameter include file as follows

```
#define REAL float
```

or

```
 #define REAL double
```

This provides a simple way to change one statement and recompile the project and see the effect upon the results. If there are significant differences, the source code must be carefully examined and some variables may need the extra precision. Most of the double precision variables were replaced with single precision variables.

Over 300 data files were evaluated with both single and double precision versions of the compare program on the laboratory computer with the following results.

Program type	Number of Files Evaluated	Number of Files with Same Results (to 5 decimal places)	Number of Files with Different Results	Maximum Numerical Difference
Double Precision	379	335	42	0.0001
Single Precision	379	335	42	0.0001

Table 2: Comparison of Single and Double Precision Calculations

A Case where Double Precision is Necessary

A rather simple case where double precision is critical is in the calculation of the standard deviation of a set of numbers. There are 2 common algorithms that can be used to compute the standard deviation. The first algorithm has the following steps:

First algorithm

1. Compute the average of the data and store it in a variable ave.
2. Then subtract the average value from all the data elements, and square the values, and take the square root of the sum.
3. $\sqrt{(1/n\sum_{i=1}^{i=n}(X_i\text{-ave})^2}$) where X_i are the data points. This formula for calculating the standard deviation doesn't usually require the use of double precision. It does however require processing the data elements twice, once to compute the average and once for computing the standard deviation. It also is very wasteful in storage since an array of size n or more is needed for storing the data points.

Second algorithm

1. Compute the average of the data as follows
2. Ave $=\sum_{i=1}^{i=n} 1/n$ (X_i)
3. Simultaneously compute the square of the data.
4. $Square = \sum_{i=1}^{i=n} 1/n$ $(X_i)^2$
5. Then the standard deviation is given by $\sqrt{}$(square-ave^2)

This method doesn't require much storage since it only uses two accumulators to sum up the data values and the square of the values, but it more susceptible to numerical errors.

Consider the following situation where the standard deviation of 500 randomly selected data points where chosen between the interval of 500 and 501.

Differences in Computing the Standard Deviation using the Two Algorithms

	First Algorithm	Second Algorithm
Single precision	0.2876	0.4231
Double precision	0.2876	0.2876

Table 3: Example of Case where Double Precision is Required

The only difference in storage was that the two accumulators for the variables in steps 2 and 4 above were in double precision instead of single precision for a difference of 8 bytes of additional storage required for the double precision data type.

Developing the code for the micro on the desktop computer

Before compiling the code with the cross compiler for the micro, it is useful to do as much of the necessary modifications as possible on the pc. The reason for this is that it is much simpler and faster to modify and test an application on the pc than the micro.

There are several stages in this effort.

Eliminating Unnecessary Routines

The compare application consists of a main program and it calls a library with 64 subroutines. The library contains many subroutines that were used by various main programs, but most of them are not needed by the compare program. For example the library had several routines to read several types of data files including ASCII and binary files. However since the micro doesn't have a file system, no actual read routines were needed. By eliminating all unnecessary routines and combining some routines, the number of required subroutines was reduced to 6. To preserve the functionality of the application it was necessary to replace a read routine with something that could get the data to the application.

The original read code was a typical routine that had as input a file name and it returned a vector of data values as output, and the number of data values. It was replaced with the following C code which emulates a read routine:

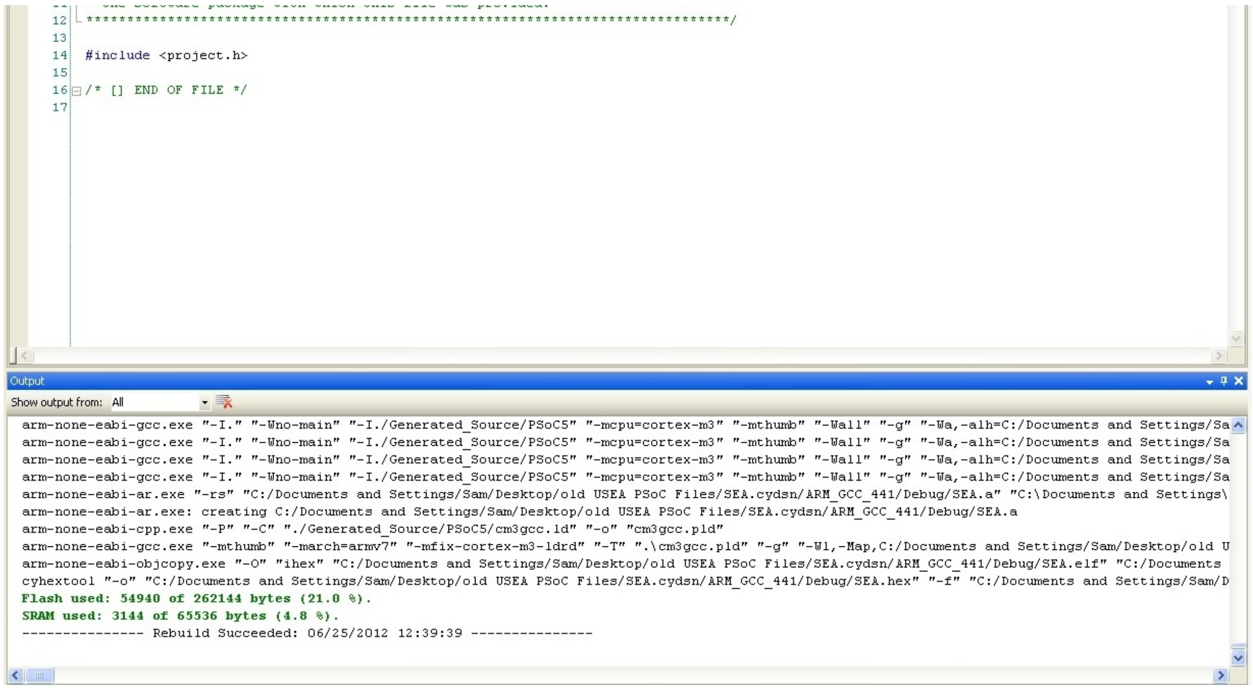
```
void ascii_read(float *Vector, int *num_data)

{

Vector[1]= 0.056880;

Vector[2]= 0.067620;

…

Vector[200]=0.041560;
*num_data=200;

}
```

	Total Number of Library Routines	Routines Added to Replace I/O Function
Laboratory Computer	64	0
Micro Computer	6	2

Table 4: Simplifying the Library

Testing the Application Designed for the Micro Computer

The revised program was compiled and executed on the pc and it gave the same results as the original program. The next step was to cross compile the program for the micro. The following image from the laboratory computer shows that micro computer version was successfully generated. The code was then uploaded to the micro where it successfully executed and gave the same numerical result as on the pc.



```
 ******************************************************************************/

#include <project.h>

/* [] END OF FILE */
```

```
arm-none-eabi-gcc.exe "-I." "-Wno-main" "-I./Generated_Source/PSoC5" "-mcpu=cortex-m3" "-mthumb" "-Wall" "-g" "-Wa,-alh=C:/Documents and Settings/Sa
arm-none-eabi-gcc.exe "-I." "-Wno-main" "-I./Generated_Source/PSoC5" "-mcpu=cortex-m3" "-mthumb" "-Wall" "-g" "-Wa,-alh=C:/Documents and Settings/Sa
arm-none-eabi-gcc.exe "-I." "-Wno-main" "-I./Generated_Source/PSoC5" "-mcpu=cortex-m3" "-mthumb" "-Wall" "-g" "-Wa,-alh=C:/Documents and Settings/Sa
arm-none-eabi-gcc.exe "-I." "-Wno-main" "-I./Generated_Source/PSoC5" "-mcpu=cortex-m3" "-mthumb" "-Wall" "-g" "-Wa,-alh=C:/Documents and Settings/Sa
arm-none-eabi-ar.exe "-rs" "C:/Documents and Settings/Sam/Desktop/old USEA PSoC Files/SEA.cydsn/ARM_GCC_441/Debug/SEA.a" "C:\Documents and Settings\
arm-none-eabi-ar.exe: creating C:/Documents and Settings/Sam/Desktop/old USEA PSoC Files/SEA.cydsn/ARM_GCC_441/Debug/SEA.a
arm-none-eabi-cpp.exe "-P" "-C" "./Generated_Source/PSoC5/cm3gcc.ld" "-o" "cm3gcc.pld"
arm-none-eabi-gcc.exe "-mthumb" "-march=armv7" "-mfix-cortex-m3-ldrd" "-T" ".\cm3gcc.pld" "-g" "-Wl,-Map,C:/Documents and Settings/Sam/Desktop/old U
arm-none-eabi-objcopy.exe "-O" "ihex" "C:/Documents and Settings/Sam/Desktop/old USEA PSoC Files/SEA.cydsn/ARM_GCC_441/Debug/SEA.elf" "C:/Documents
cyhextool "-o" "C:/Documents and Settings/Sam/Desktop/old USEA PSoC Files/SEA.cydsn/ARM_GCC_441/Debug/SEA.hex" "-f" "C:/Documents and Settings/Sam/D
Flash used: 54940 of 262144 bytes (21.0 %).
SRAM used: 3144 of 65536 bytes (4.8 %).
--------------- Rebuild Succeeded: 06/25/2012 12:39:39 ---------------
```

Figure 5: Output from the Cross Compiler for the Micro

As is shown in Figure 5, only a small amount of Flash and SRAM resources were used for this application. The next question is will the database builder application fit and run on the micro?

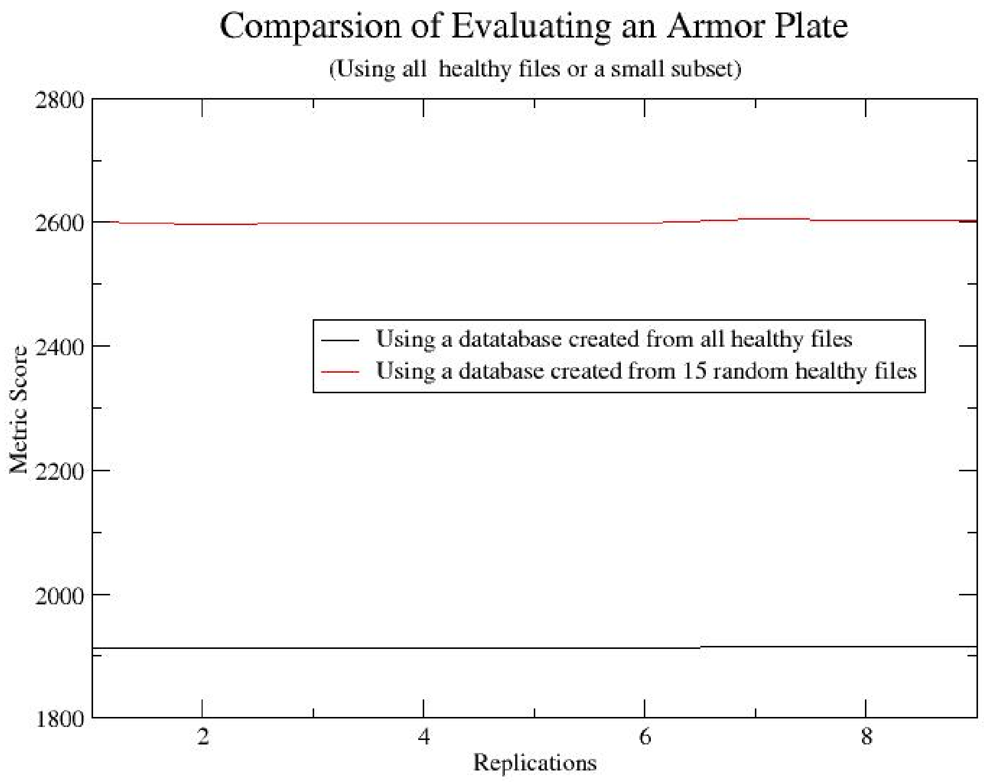
The More Complicated Main Program: The Database Builder

This program is called the fingerprint program because the database is sort of like a "fingerprint" of the armor panel that it represents. One of the principles of statistical theory is that by repeating an experiment over time, random errors will tend to cancel out each other. This requires the collection of data over time and under varying ambient conditions. Collecting and storing day over time requires the existence of a file system and a clock, but not an operating system. Fortunately a file system is commercially available for the micro that only requires 1300 bytes of storage. The data can be stored on a compact flash or sd card either of which can be integrated into the system hardware. To build a representative fingerprint database can require the collection of several hundred data files. The compact flash card has more than enough space to store these data files since each data file contains only 804 bytes of data. One thousand data files would only require 0.8 megabytes of data, and a common compact flash card can hold 32 GB of data. However the micro doesn't have sufficient storage to hold this data in core for processing. The solution to this problem is to use random sampling and choose a small subset of the total number of data files for processing. The following table compares the results of using the original program and the one that was rewritten for the micro.

Program used in creating database	Average Value	Standard Deviation
Original (150 files)	7.47*10-3	3.08*10-3
Micro version (15 files)	9.79*10-1	2.89*10-3

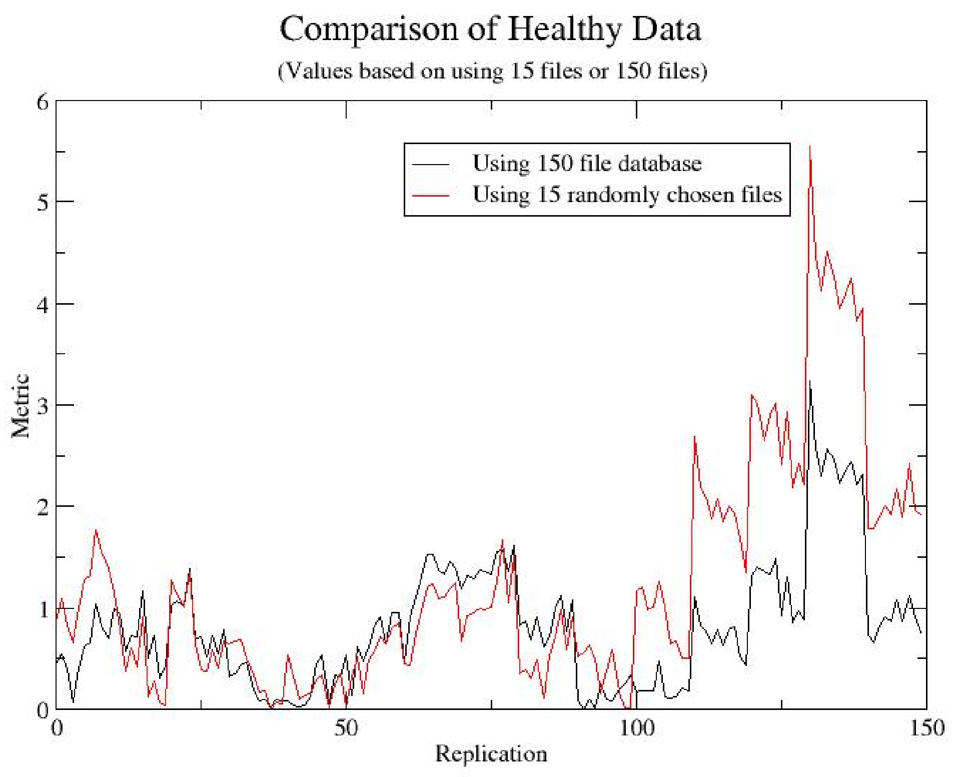
Figure 6: Comparison of the main database values from the 2 versions of the program

The table above compares the database values from the two versions of the program, but it doesn't really give much of an idea as to the ability of the micro version to produce results that are comparable to the original version. A better way to compare the two versions is to see graphically depict the metric values using both of the two databases. Figure 7 shows this comparison. In general the values are similar, although some of the values using the micro version are twice as large as those for the original version. The really critical test however, is can the micro version detect changes in armor health as well as the original version? Figure 8 shows that both versions are able to detect changes with similar precision. In fact the scores from the micro version are slightly higher than those for the original version. The average score for the original version is slightly greater than 1900, while the score for the micro version is about 2600.



Comparsion of Evaluating an Armor Plate
(Using all healthy files or a small subset)
2800
2600
2400
2200
2000
1800
Metric Score
2
4
6
8
Replications
Using a datatabase created from all healthy files
Using a database created from 15 random healthy files

Trial Number	Using database derived from 150 files	Using database derived from 15 files
1	1911.88	2600.283
2	1911.55	2597.47
3	1912.12	2598.15
4	1912.25	2598.04
5	1913.52	2599.72
6	1911.57	2597.90
7	1915.28	2604.73
8	1914.57	2603.23
9	1915.03	2603.42
10	1914.34	2602.11


Comparison of Healthy Data
(Values based on using 15 files or 150 files)
Using 150 file database
Using 15 randomly chosen files
Metric
Replication
0
1
2
3
4
5
6
0
50
100
150